Multi software product lines in the wild

Multi Software Product Lines in the Wild

Michael Lienhardt
michael.lienhardt@unito.it
Università di Torino
Italy

Ferruccio Damiani
ferruccio.damiani@unito.it
Università di Torino
Italy

Simone Donetti
simone.donetti@unito.it
Università di Torino
Italy

Luca Paolini
luca.paolini@unito.it
Università di Torino
Italy

ABSTRACT

Modern software systems are often built from customizable and inter-dependent components. Such customizations usually define which features are offered by the components, and may depend on backend components being configured in a specific way. As such system become very large, with a huge number of possible configurations and complex dependencies between components, maintenance and ensuring the consistency of such systems is a challenge.

In this paper, we propose a Multi Software Product Line model to capture the complexity of such systems and pave the way to formal studies on them. We applied and implemented our model on a full Linux Distribution of almost 40,000 interconnected components and 3 million features, and present some initial analysis we did on this model.

CCS CONCEPTS

• **Software and its engineering → Software design engineering**; **Software product lines**; *Feature interaction*; *Abstraction, modeling and modularity*; *Software libraries and repositories*; *Software creation and management*;

KEYWORDS

Software Product Line, Multi Software Product Line, Configurable Software, Variability Modeling, Composition, Linux Distribution

ACM Reference Format:
Michael Lienhardt, Ferruccio Damiani, Simone Donetti, and Luca Paolini. 2018. Multi Software Product Lines in the Wild. In *VAMOS 2018: 12th International Workshop on Variability Modelling of Software-Intensive Systems, February 7–9, 2018, Madrid, Spain,* Malte Lochau (Ed.). ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3168365.3170425

This research has been partially funded by: EU Horizon 2020 project HyVar (www.hyvar-project.eu), GA No. 644298; ICT COST Action IC1402 ARVI (www.cost-arvi.eu); Ateneo/CSP D16D15000360005 project RunVar (runvar-project.di.unito.it);

VAMOS 2018, February 7–9, 2018, Madrid, Spain

ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3170425

1 INTRODUCTION

A *Software Product Line* (SPL) is a set of similar programs, called *variants*, with a common code base and well documented variability [1, 6, 19]. Modern software systems are often built as complex assemblages of customizable *components* that out-grow the expressiveness of SPLs. Consider for instance a core wordpress web server. Such system is built from at least five components: i) the actual wordpress php code; ii) a web server; iii) a php interpreter; iv) a database to host the wordpress data; and v) the data in the database. Interestingly, many of these components can be realized by different software tools: the most common choice for the web server and for the database is *apache* and *mysql*, but other options (like *nginx* and *berkley-db*) are possible. Moreover, these components are customizable in order to satisfy various functional and non-functional requirements of the users. For instance, apache can support many authentication protocols, many scripting languages (for which it may required a backend interpreter) which may be activated or not by the user, depending on its requirements. In our example, apache must have its php support enabled in order to execute wordpress, which requires a php interpreter, but activating more of its features may just add to its memory and computation load without any benefit for the overall system.

In such composed systems the concept of SPL can be used to describe each customizable software individually, while additional mechanisms are needed to describe the relationship between different SPLs (either *requirements* like in our example where apache requires a php interpreter, or *conflicts* where two SPLs cannot be used together) and to describe the concept of *components* that can be realized by different SPLs with similar functionalities. Therefore these systems can be described as *Multi Software Product Lines* (MPLs): an MPL is a sets of interdependent SPLs that need to be managed in a decentralized fashion by multiple teams and stakeholders [15].

In this paper, we build upon previous works [10, 22] to define a formal model of MPL that aims to: i) capture the notion of relationship between different components; ii) be flexible enough to describe different real composed systems; and iii) support the construction of tools to help in designing, maintaining and analyzing composed systems. More precisely, the contributions of this paper are:

- a formal definition of SPL agnostic on a number of implementation details, so it encompasses SPLs built with different SPL approaches (we refer to [21] and [26] for a survey of

different approaches for implementing and analyzing SPLs, respectively);
- a formal definition of MPL based on the notions of: *Dependent SPL* which extends SPLs with *dependencies* capable of expressing both requirements and conflicts; and *SPL subtyping* which describes the commonalities shared between different SPLs, thus capturing the concept of components;
- an implementation of this MPL model on top of the Linux *Gentoo* distribution [13];
- a use of this implementation to extract interesting statistical data on how MPLs are used in practice; and
- a discussion on the benefits of using a formal MPL model in designing such large collection of components.

The rest of the paper is structured as follows: Section 2 constructs step by step our MPL model; Section 3 introduces *portage*, the package manager of the *Gentoo* Linux distribution, and shows how it can be mapped into our model; Section 4 presents our analysis on portage and its MPL structure while Section 5 discusses some limitations of portage and how our model, together with some extensions, could help in designing, maintaining and analysing composed systems; finally, Section 6 discusses related work and Section 7 concludes the paper.

2 MPL MODEL

To motivate an illustrate our model, we use a running example based on the main use case of the EC H2020 project HyVar which models software systems in a car. This use case is structured in three *Electronic Control Units* (ECUs) with their dedicated functionalities

- *ECU_A* is responsible for the core functionalities of the car, which include i) the mandatory *emergency call* that automatically calls the police in case of a crash using either the european union (EU) or the russian protocol; and ii) an optional *connnection Gateway* that allow message exchanges between different ECU in the car.
- *ECU_B* hosts the optional services of the car, that include i) a *gear advice* that hints the driver in when switching gear; and ii) a *brake advice* that gives useful information about the brake status.
- *ECU_C* hosts the UI of all the services of the car.

Each of these ECUs are implemented by their dedicated teams, using different implementation languages and variability approaches to encode the ECU customization. Hence, for our model to be able to capture this use case, its notion of variant must be able to describe any kind of structure, may it be code, library, data, etc. One important property of variants however is that they can be *composed* in order to buid up composed systems, like in our car use case, which is composed of three ECUs. Note however that not every variants can be composed together, e.g., two java jar files could declare the same class. We thus get the following definition for variants:

Definition 2.1 (Variants). The set of all variants $\mathcal{V}$ is a set of software components equipped with a structure of a *partially commutative monoid* (PCM) [12, 27], i.e., a triple $(\mathcal{V}, \oplus, \varepsilon)$ where $\oplus$ is a partial and commutative *composition* operator and ε is its neutral element. More precisely, writing $x \perp y$ when $x \oplus y$ is defined, we have the following properties:

(1) $x \perp y$ implies $y \perp x$ and $x \oplus y = y \oplus x$;
(2) $y \perp z$ and $x \perp (y \oplus z)$ imply $x \perp y$, $(x \oplus y) \perp z$ and $x \oplus (y \oplus z) = (x \oplus y) \oplus z$;
(3) $\varepsilon \perp x$ and $\varepsilon \oplus x = x$.

Our definition of Software Product Line build upon this abstract notion of variant, to which it adds variability (or customization) using a *feature model* and a *generator function* that maps the different products of the feature model to its corresponding variant:

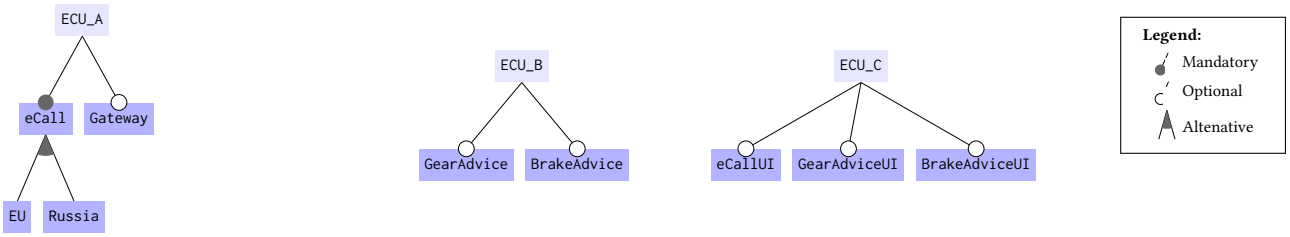
Definition 2.2 (Feature Model, Software Product Line). A *Feature Model* $\mathcal{M}$ is a pair $(\mathcal{F}, \mathcal{P})$ where $\mathcal{F}$ is a set of features and $\mathcal{P} \subseteq 2^{\mathcal{F}}$ is a set of products. $\mathcal{M}_\emptyset = (\emptyset, \emptyset)$ is the *empty* feature model.

A *Software Product Line* $\mathcal{L}$ is a pair $(\mathcal{M}, \mathcal{G})$ where $\mathcal{M}$ is the feature model of the SPL and $\mathcal{G}$ is the generator of the SPL, i.e., a partial function from the products of $\mathcal{M}$ to variants $v \in \mathcal{V}$.

Note that in this definition, the generator of the SPL $\mathcal{G}$ can be partial, i.e., some product may not have a corresponding variant, to also capture SPLs that are ill-defined and contains errors that make impossible the generation of a product's variant. Note also that our notion of generator does not specify how variants are generated from a specific product. This allows our model to capture several SPL approaches, like annotative product lines [1] (e.g., where the generator works by applying the C preprocessor on some source code), Feature-Oriented Programming [1] (where the generator works by combining the artifacts stored in the selected features' modules), or Delta-Oriented Programming [20] (where the generator works by applying selected delta modules on a intial core artifact).

Feature models can be graphically represented as *feature diagrams*, arranging the features in a tree structure with additional *cross-tree constraints* to describe their dependencies (see, e.g., [3]). Figure 1 presents the feature models of the different ECUs in our running example. As previously discussed, ECU_A (in Figure 1a) has a mandatory feature called `eCall` implementing the emergency call procedure that calls the police in case of a car crash, and that has two aternative sub-features corresponding to the two possible call protocols. Additionally ECU_A has one additional optional feature called `Gateway` that implements communication between the different ECUs in the car. ECU_B on the other hand (in Figure 1b) has two optional features: `GearAdvice` that corresponds to the *gear advice* service, and `BrakeAdvice` that corresponds to the *brake advice* service. Finally, ECU_C has one optional feature per possible user interface: `eCallUI` corresponds to the UI of the emergency call functionality (implemented in ECU_A), `GearAdviceUI` corresponds to the UI of the gear advice service and `BrakeAdviceUI` corresponds to the UI of the brake advice service (both services being implemented in ECU_B).

While the feature models in Figure 1 describe the variability of each ECU of the car, they do not express their intrinsic relationships. For instance, for the feature `GearAdviceUI` (in ECU_C) to make sense, the gear advice service must be present in ECU_B (so the data to put in the user interface would be computed), and the communication between the ECUs (implemented in ECU_A) must be available (so the data computed in ECU_B could be sent to ECU_C). Hence, in this example, the features of ECU_C *depend* on the activation of some features in ECU_A and ECU_B.



(a) Feature Model of ECU_A

(b) Feature Model of ECU_B

(c) Feature Model of ECU_C

Figure 1: Feature Models of the ECUs

To capture such dependencies, we consider a notion of *Dependent SPL* (DPL) that extends the concept of SPL with an explicit notion of dependency. We illustrate this notion with our ECU_C example: this SPL thus must be extended to explictly state that: i) it depends on some features of ECU_A and ECU_B; and ii) how precisely its own features are related to the ones of ECU_A and ECU_B. Syntactically, the ECU_C DPL could be written as in Listing 1 below: the dependencies toward ECU_A and ECU_B are explicitly stated with the keword **depends on**, while the three constraints at the end of the declaration precisely describe the dependencies between the features of ECU_C and the service implemented in ECU_A and ECU_B.

Line ECU_C:
depends on ECU_A
depends on ECU_B
eCallUI ⇒ECU_A.eCall ∧ECU_A.Gateway
GearAdviceUI ⇒ECU_B.GearAdvice ∧ECU_A.Gateway
BrakeAdviceUI ⇒ECU_B.BrakeAdvice ∧ECU_A.Gateway

Listing 1: Declaration of the ECU_C DPL

Formally, the **depends on** syntax corresponds to the ECU_C's feature model *including* the features of ECU_A and ECU_B so it can relate to them, while the constraint syntax specify the structure of the ECU_C's products that encodes the dependencies between the features of the different ECUs, in the same way as the product of an SPL encodes the dependencies between the features within the SPL. Our definition is based on a notion of *feature model extension* that derives from the more complex notion of *feature model composition* presented in [22].

Definition 2.3 (Extension of a FM, Dependent SPL). A feature model $\mathcal{M} = (\mathcal{F}, \mathcal{P})$ is an *extension* of a feature model $\mathcal{M}' = (\mathcal{F}', \mathcal{P}')$, written $\mathcal{M}' \hookrightarrow \mathcal{M}$, iff $\mathcal{F}' \subseteq \mathcal{F}$ and for all $p \in \mathcal{P}$, there exists $p' \in \mathcal{P}' \cup \{\emptyset\}$ such that $p \cap \mathcal{F}' = p'$.

A *Dependent SPL* $\mathcal{L}$ is a triple $(\mathcal{M}, \mathcal{G}, \mathcal{D})$ where $(\mathcal{M}, \mathcal{G})$ is an SPL and $\mathcal{D}$ is a set of dependencies, i.e., a set of DPLs $\{\mathcal{L}_i = (\mathcal{M}_i, \mathcal{G}_i, \mathcal{D}_i)\}_{i \in I}$ such that $\mathcal{M}_i \hookrightarrow \mathcal{M}$ for all $i \in I$.

Note that every SPLs $(\mathcal{M}, \mathcal{G})$ can seamlessly be extended into the DPL $(\mathcal{M}, \mathcal{G}, \emptyset)$: in the rest of the document, we will consider that an SPL is a DPL with an empty set of dependencies. Hence, in our example, ECU_A and ECU_B are indeed DPLs, and ECU_C does correspond to this definition. In particular, it is the constraint $\mathcal{M}_i \hookrightarrow \mathcal{M}$ that enforces that the features of ECU_A and ECU_B are correctly transfered in the ECU_C DPL in our example.

The above definition of Dependent SPL is strongly-coupling: a DPL $\mathcal{L}$ depends on a set of specific DPLs, and if one of them is replaced by a new version, then $\mathcal{L}$ must be changed to update its dependencies. Moreover, such a strongly coupling forbids the designs of systems as discussed in the introduction, where a *web server* component (i.e., dependency) could be filled by several equivalent SPLs. To solve this issue, we consider a notion of *subtyping*. Namely, we assume that some SPLs do not actually generate concrete code (like interfaces in Java 7), and that there is a subtyping relation that allow to establish whether a *concrete SPL* implements an *abstract SPL*. This subtyping relation must validate some consistency properties. In particular, variability must be preserved by subtyping: if the DPL $\mathcal{L}$ is a subtype of $\mathcal{L}'$, then all the products of $\mathcal{L}'$ must be extensible into a product of $\mathcal{L}$. This property, inspired by the notion of *feature model interface* defined in [22], enables a DPL depending on another DPL $\mathcal{L}'$ to seamlessly use the products of $\mathcal{L}$ to resolve its dependencies. With this, we can give the definition of a Multi Software Product Line:

Definition 2.4 (Concrete/Abstract DPL, Refinement of an FM, MPL). Consider the set of variants $\mathcal{V}$ to be partitioned into two subsets: the set of *concrete* variants $\mathcal{V}_c$ and the set of *abstract* variants $\mathcal{V}_a$. A DPL $(\mathcal{M}, \mathcal{G}, \mathcal{D})$ is *concrete* if it generates only concrete variants: $\text{im}(\mathcal{G}) \subset \mathcal{V}_c$. A DPL $(\mathcal{M}, \mathcal{G}, \mathcal{D})$ is *abstract* if it generates only abstract variants: $\text{im}(\mathcal{G}) \subset \mathcal{V}_a$.

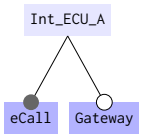
A feature model $\mathcal{M} = (\mathcal{F}, \mathcal{P})$ is a *refinement* of another feature model $\mathcal{M}' = (\mathcal{F}', \mathcal{P}')$, written $\mathcal{M} \triangleright \mathcal{M}'$ iff $\mathcal{F}' \subseteq \mathcal{F}$ and for all $p' \in \mathcal{P}'$, there exists $p \in \mathcal{P}$ with $p \cap \mathcal{F}' = p'$.

An MPL is a pair $\mathcal{K} = (\mathcal{S}, \preceq)$ where $\mathcal{S}$ is a set of concrete and abstract DPLs and $\preceq$ is a subtyping relation between the DPL in $\mathcal{K}$ ($\preceq \subset \mathcal{S} \times \mathcal{S}$) that validates the following properties:

(1) if $\mathcal{L}$ is concrete, there exists no $\mathcal{L}'$ such that $\mathcal{L}' \preceq \mathcal{L}$
(2) if $\mathcal{L}$ is abstract, there exists $\mathcal{L}'$ with $\mathcal{L}' \preceq \mathcal{L}$
(3) $\mathcal{L}_1 \preceq \mathcal{L}_2$ and $\mathcal{L}_2 \preceq \mathcal{L}_3$ implies $\mathcal{L}_1 \preceq \mathcal{L}_3$
(4) $(\mathcal{M}, \mathcal{G}, \mathcal{D}) \preceq (\mathcal{M}', \mathcal{G}', \mathcal{D}')$ implies $\mathcal{M} \triangleright \mathcal{M}'$

In our car example, the MPL is simply the collection of the three ECU_A, ECU_B and ECU_C DPLs, with an empty subtyping relation. It would however make sense to add an abstract SPL on top of the ECU_A: indeed, per design, this ECU is closely bound to the architecture of the car, and so, if in the future other car models would be supported, new ECU_A DPLs would have to be implemented. Our MPL thus becomes the collection of four DPLs: ECU_A, ECU_B, ECU_C and the new Int_ECU_A DPL with ECU_A $\preceq$ Int_ECU_A,

ECU_C now depending on Int_ECU_A instead of ECU_A, and the feature model of Int_ECU_A being:



Note that the feature model of Int_ECU_A does not include the features `EU` and `Russia` of ECU_A: these two features are not used by ECU_B nor ECU_C, and our definition of *refinement* allows for such a simplification. Such flexibility also means that other implementations of ECU_A could also have a feature model rather different from the current one, and could still be useable by ECU_C as long as they have a mandatory feature `eCall` and an optional feature `Gateway`.

The last part of our model concerns the variant generation within an MPL. With the added notion of dependency, generating a variant in an MPL does not consists of only choosing the product of a DPL and applying its generator $\mathcal{G}$ on it: we need to consider the DPL's dependencies, choose a product and generate a variant for each of them, and combine all these variants into a complete one. Additionally, the chosen products for the dependencies must validate the constraints specified in the DPL: e.g., when generating a variant for ECU_C with the feature `GearAdviceUI` selected, the corresponding variant of ECU_A must have the feature `Gateway` implemented. Hence, a variant generation within an MPL is not defined by only one product, but by what we call a *multi-product* which states how every DPL of the MPL is used in the MPL variant generation. More precisely, a multi-product m is a partial function from the DPLs of the MPL to one of their product, where: i) the domain of m defines which DPLs are used in the generation of that multi-product's variant; ii) $m(\mathcal{L})$ is the product of $\mathcal{L}$ which is used in the generation of that multi-product's variant; and iii) the products in the image of m must be consistent w.r.t. the constraints specified in the different DPL as previously discussed. This notion of multi-product is formalized as follows:

Definition 2.5 (Multi-product, Generator of an MPL). A *multi-product* of an MPL $\mathcal{K} = (\mathcal{S}, \preceq)$ is a partial function m from the DPLs of $\mathcal{K}$ to one of their respective product such that:

(1) $\mathcal{L} = (\mathcal{M}, \mathcal{G}, \mathcal{D}) \in \mathrm{dom}(m)$ implies that:

$$\forall \mathcal{L}' = ((\mathcal{F}', \mathcal{P}'), \mathcal{G}', \mathcal{D}') \in \mathcal{D}, \mathcal{L}' \in \mathrm{dom}(m) \wedge m(\mathcal{L}) \cap \mathcal{F}' = m(\mathcal{L}')$$

(2) $\mathcal{L} = ((\mathcal{F}, \mathcal{P}), \mathcal{G}, \mathcal{D}) \in \mathrm{dom}(m)$ and $R = \{\mathcal{L}' \mid \mathcal{L}' \preceq \mathcal{L}\} \neq \emptyset$ implies that $R \cap \mathrm{dom}(m) \neq \emptyset$ and:

$$\forall \mathcal{L}' \in R \cap \mathrm{dom}(m), m(\mathcal{L}') \cap \mathcal{F} = m(\mathcal{L})$$

The *generator* $\mathcal{G}_{\mathcal{K}}$ of an MPL $\mathcal{K}$ is a partial function from the multi-products of $\mathcal{K}$ defined as follows:

$$\mathcal{G}_{\mathcal{K}}(m) = \begin{cases} \bigoplus_{\mathcal{L}=(\mathcal{M},\mathcal{G},\mathcal{D}) \in \mathrm{dom}(m)} \mathcal{G}(m(\mathcal{L})) & \text{if defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

To illustrate this definition of multi-product, suppose we want to generate the variant of a car with the features `EmergencyCallUI` and `GearAdviceUI` of ECU_C activated. A possible multi-product that validates this selection could be m defined as follows:

$$\begin{cases} m(\texttt{ECU\_C}) = \texttt{eCallUI}, \texttt{GearAdviceUI}, \\ \qquad\qquad\quad \texttt{Int\_ECU\_A.eCall}, \texttt{Int\_ECU\_A.Gateway} \\ m(\texttt{ECU\_B}) = \texttt{GearAdvice} \\ m(\texttt{Int\_ECU\_A}) = \texttt{eCall}, \texttt{Gateway} \\ m(\texttt{ECU\_A}) = \texttt{eCall}, \texttt{Gateway}, \texttt{EU} \end{cases}$$

3 PORTAGE PACKAGE MANAGER

Gentoo [13] is a Linux distribution focused on optimization and customization. Like many Linux distributions, the core of Gentoo is its package manager, called *Portage* [14], that eases the installation and management of software on the computer. Unlike most package managers, Portage is a *source-based* manager, i.e., installing a package with portage consists in downloading the source code of the software, compiling and installing it locally. This approach allows for the compiled packages to use all the functionalities of the host hardware (thus enabling optimization), but also to be *customized* by the user who can select, during the compilation process, which features he wants installed in the software. In that sense, each package in portage is an SPL, and the full package repository of Portage forms a collection of SPLs, i.e., an MPL. We illustrate in this Section how our model captures the MPL structure of Portage's repository.

Like most package managers, Portage's packages are specific versions of standard software, like `apache-2.2.32` or `antlr-4.5.1`, developed by their own teams. Consequently, the actual implementation language of each package, together with its variability are the responsability of that software's development team. Most softwares are implemented in C or C++ and use the preprocessor's `#ifdefs` to encode variability, with a *configure* script to select the features to compile; but projects based on another programming language can use a different compilation mechanism to implement variability. One of the main functionality of Portage is to offer a unified layer on top of the specific implementation of each package that captures all the important aspects of package configuration and installation. This unified layer is in most part defined by a set of *.ebuild* files, one per package, each of them containing the following information:

- the feature model of the package, declared with a list of features (called *USE flags* in Portage) and an additional constraint that specifies which features can be selected together;
- the dependencies of the package, declared in a similar fashion as the constraints in Listing 1 (the **depends on** statements are implicit in Portage); and
- the generator function of the package, defined with a set of different scripts relating the feature selection to the compilation process of that package.

Portage supports modularity by means of *atoms*: instead of referencing a specific SPL in a constraint (like we did in Listing 1), Portage allows to use atoms, i.e., a kind of pattern that can be resolved in more than one SPL. We illustrate our description of Portage with Listing **??** which shows an excerpt of the .ebuild file corresponding to the package of the version `16.02-r1` of the `p7zip` archiver.

```
KEYWORDS="alpha amd64 ~arm hppa ia64 ppc [...]"
IUSE="abi_x86_x32 amd64 x86 doc kde pch rar static wxwidgets"
REQUIRED_USE="kde? ( wxwidgets )"
```

```
DEPEND="wxwidgets? ( x11-libs/wxGTK:3.0[X] )
        abi_x86_x32? ( >=dev-lang/yasm-1.2.0-r1 )
        amd64? ( dev-lang/yasm )
        x86? ( dev-lang/nasm )"
src_prepare() { [...] }
src_compile() { [...] }
src_install() { [...] }
```

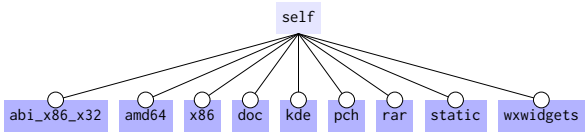
Listing 2: Excerpt of the p7zip-16.02-r1 Package

In this example, **KEYWORDS** lists the hardware architectures on which the package can be installed (we trucated the list in our example, as the full list is long), **IUSE** lists the features of this package, **REQUIRED_USE** describes how features can be selected together and **DEPEND** is the constraint defining the dependencies of this package. Additionally, the three functions src_prepare, src_compile and src_install implement the generator of the package, specifying respectively how to prepare the source code, how to compile it and how to install the resulting variant on the system.

The constraint in the **REQUIRED_USE** variable states that selecting the kde feature requires also selecting wxwidgets. The dependencies listed in the **DEPEND** variable is structured in three constraints. The first one states that if the wxwidgets feature is selected, then a package implementing the atom x11–libs/wxGTK:3.0 must be also installed. Moreover, the [X] syntax means that this package must be installed with the feature X selected. The second constraint states that if the abi_x86_x32 feature is selected, then a version greater or equal to `1.2.0-r1` of the `yasm` program must be installed, while the third line does not give any restriction on the version of `yasm` that must be installed in case the feature amd64 is selected. Finally, the x86 feature requires any version of `nasm` to be installed.

Portage can be encoded in our model in the following way: all Portage's packages are concrete SPLs, while atoms are abstract SPLs without any variant but with a set of concrete SPLs implementing them: the matching function between an atom and a package in Portage corresponds to our model's subtyping relation. For instance, the `p7zip-16.02-r1` package can be encoded as described in Listing **??**, where the root feature of any SPL is called `self`.

Line p7zip–16.02–r1:



depends on x11–libs/wxGTK:3.0
depends on >=dev–lang/yasm–1.2.0–r1
depends on dev–lang/yasm
depends on dev–lang/nasm
kde ⇒wxwidgets
wxwidgets ⇒"x11–libs/wxGTK:3.0".X
abi_x86_x32 ⇒">=dev–lang/yasm–1.2.0–r1".self
amd64 ⇒"dev–lang/yasm".self
x86 ⇒"dev–lang/nasm".self

Listing 3: Declaration of the p7zip-16.02-r1 DPL

We invite the interested reader to look at [17] for more details on Portage and its MPL structure.

4 EXPLORING PORTAGE

Due to its size and its large user and developer communities, we believe that Portage could be a valuable source of information on how MPLs are used in the wild. We thus implemented a prototype version of our model, together with an importer that extracts the MPL structure from Portage and analysis tools that compute some information from that structure. For our analysis, we considered the `Gentoo 201703 (CLI Minimal)` version of the `osboxes` Gentoo Virtual Machine[1].

This version of Portage contains 38907 concrete SPLs and 31264 abstract SPLs. The feature model of a concrete SPL has in average 71.75 features. However this number is artificially large because Portage adds many (between 68 and 191) hardware-related features to all concrete SPLs, even those that do not use them. Our estimation (looking at the constraints in the feature model and at the generator function) is that only 4 or 5 features are actually used per concrete SPL in average. Interestingly, most concrete SPLs (30038) have a very simple feature model where all the features are optional and so most concrete SPLs have between 16 and 32 products. Consequently, we can estimate an over-approximation of the number of multi-products for portage: considering that we must choose one product per SPL (the SPL not being installed corresponding to the empty product), and considering an SPL to have 24 products (i.e., between 16 and 32), we obtain an approximation of 24^{39807} multi-products. Note that in Portage, abstract SPLs do not have explicit feature models: they implicitly inherit the feature models of the concrete SPLs that implement them. Moreover, as their configuration is closely related to the configuration of their concrete SPL implementations, we did not consider them in our analysis.

The second part of our analysis focuses on the *dependencies* and the *subtyping relation* in Portage. To do so, we constructed a directed *dependency graph* where the nodes are the SPLs (concrete and abstract) of Portage, and where an edge from a concrete SPL to an abstract SPL corresponds to a dependency, while an edge from an abstract SPL to a concrete one corresponds to the subtyping relation (in portage, abstract SPL do not have dependencies). Based on this graph, we obtained that in average a concrete SPL has 9 dependencies, of which 3 are conditional and 2.2 requires specific features to be selected or unselected in the abstract SPL. On the other hand, abstract SPLs have in average 2.2 concrete SPLs that implement them. We moreover noted that 2380 abstract SPLs do not have any implementation: these abstract SPLs are mostly used to declare possible conflicts to packages that do not exist anymore. The dependency graph itself does not have a specific structure: it is not connected, with 935 concrete SPLs that do not implement any abstract SPL and without any dependencies; and it is not acyclic, as it contains 113950 loops, with its biggest *strongly connected component* containing 3009 concrete SPLs. Most of the lone concrete SPLs are simple data or utility packages, like fonts (`konfont-0.1`) or compression tools (`lz5-2.0`). On the other hand, while part of the loops in Portage encode conflicts between versions, the motivation for most of them and their size remains unknown.

Finally, we developed a visualization tool prototype, based on `tulip` [2], to display parts of the Portage dependency graph. The

[1] Available at `http://www.osboxes.org/gentoo`

main challenge in designing this tool was the size of the data to display: with concrete SPLs having 9 dependencies in average, each of them having 2.2 possible implementations, displaying just a part of the dependency graph would include several hundred nodes. To solve this issue, our tool uses two abstractions. First, it removes the abstract SPLs, combining the edges so the concrete SPLs point directly the implementations of their dependencies. Second, it merges the concrete SPLs that correspond to different versions of the same software. Indeed, these SPLs have a very similar, if not equal feature model and dependency set, so it is sound to unify them, allowing in the process to reduce by 2 the size of the displayed graph. Figure 2 presents two sets of SPLs: on the left are shown all the SPLs of the `kde-frameworks` category, while on the right are shown all the SPLs of the `gnome-base` category. These two sets of SPLs are interesting to display as they are an important example of complex softwares that is still of a manageable size (`kde-frameworks` contains 160 concrete SPLs while `gnome-base` contains 66 concrete SPLs). The size and color of the nodes are defined by how many SPLs depends on them, transitively. On the `kde` side, we thus see clearly that `kdelibs`, which implements all the core functionalities of `kde`, is indeed a core SPL in this graph. On the `gnome` side, it is the `gnome` and `gnome-extra-apps` that are the main SPLs in this set. Another important property of these graphs is that they are far more dense (0.08) than the complete dependency graph ($6 \cdot 10^{-3}$). Indeed, the original images were unreadable due to the number of edges between the different SPLs: this was solved by bundling the edges.

5 DISCUSSION

The preliminary results of the application of our model on Portage suggest that most of the MPL structure of Portage can be encoded in our model. For instance, important features of Portage's packages, like their feature models or their dependencies, can be easily captured in our model.

There are, however, some crucial differences between our MPL model and the MPL structure in Portage. For instance, like many other package managers, Portage defines several installation steps for its packages, each of them with its specific set of dependencies. On the opposite, our model does not consider installation steps, and our SPLs have only one set of dependencies. Consequently, when we imported the Portage MPL in our model to perform our analysis, we had to merge together all the dependencies of each package. On the other hand, the notion of atom in Portage is quite weak compared to the notions of dependency and subtyping in our model. While atoms are easy to manipulate (writing one atom implicitly declares an abstract SPL and its subtyping relation), they are not expressive enough to directly capture our wordpress example in the introduction: it is impossible in Portage to declare an atom that corresponds to any implementation of a web-server. Portage partially circumvents this issue using *virtual* packages, i.e., declaring concrete SPLs in place of abstract ones, and using dependencies instead of subtyping. Moreover, the capability of our abstract SPL to hide away parts of the feature model of its implementations is not present in Portage. This limitation of Portage's atoms is actually natural: hiding, i.e., defining a clear an consistent API for a software, is out of the scope of Portage as its goal is to offer a unified interface for the variability of all of its packages; instead, hiding is a concern for the teams that implement the packages. Unfortunately, the fact that atoms do not have an explicit feature model, and the difficulty to understand the modeled subtyping relation is a cause for many bugs in Portage. This problem, as well as some other issues in Portage's design are clearly visible on its bug-tracker[2]. While many bugs in Portage are compilation errors, due to a problem in the package's generator function, many other bugs are caused by the difficulties to define correct atoms, correct feature models (e.g., bugs `578658` and `517252`), and to analyse them. In some cases, a package cannot be installed and the user does not understand why due to the unclear semantics of the constraints used to define the feature model. In other cases, the default configuration of the package (written by the package's maintainer) is not a valid product of the feature model (e.g. bug `607360`). Some other bugs are caused by wrongly written atoms that declare dependencies that do not have any implementation (e.g., bugs `360019` and `618540`). Also, the unclear semantics of atoms is the cause of several bugs in the dependency resolver of Portage itself (e.g., bugs `528836` and `608546`).

We believe that making the notions of atoms and constraints in Portage more formal could help in avoiding some of the bugs in its package definitions. Moreover, having a more formal notion of atoms and constraints in Portage could lead to a declaration of a feature model that could be analyzed by existing tools, like [4, 18, 24]. Finally, adopting a more formal notion of constraints for its feature model implementation could help Portage to use a more robust off-the-shelf constraint solver as back-end to its dependency resolver, thus making that central part of Portage less error-prone.

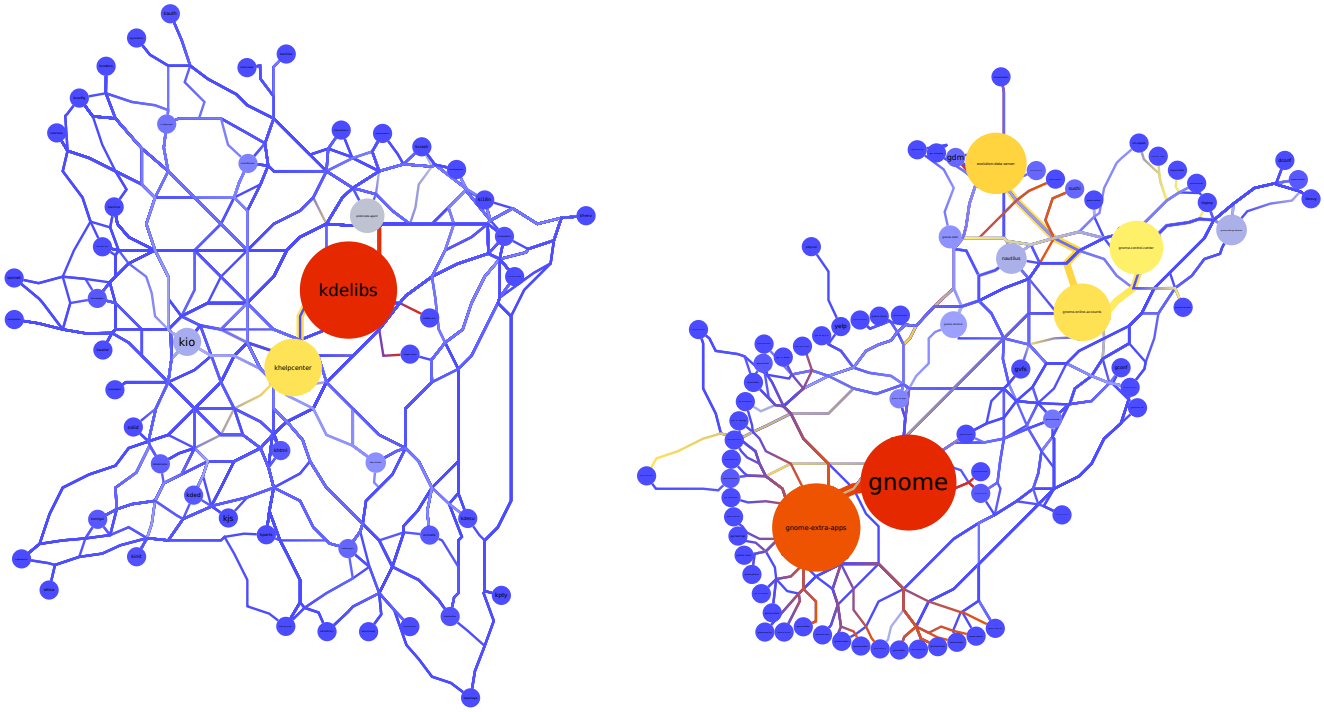
6 RELATED WORK

This current paper is related to previous work [10] that introduced a similar MPL structure on top of the *Delta-Oriented Programming* (DOP) [20] approach for implementing SPL. The MPL structure proposed in this paper however differs in various points from [10]. It is more general, as we use a generic notion of variant instead of a specific programming language, and we also have a generic notion of SPL instead of a specific implementation of the concept. It is also more flexible, as we replaced its modularity mechanism, based on *SPL Signatures* and *Implementation relation*, with a more flexible notion of subtyping that allows for more freedom in how dependencies are resolved and for *refinement* between declarative SPLs. On the other hand, [10] constructs a theory of compositional analysis of MPL that is not present in this current work.

A different approach for defining MPL on top of DOP has been outlined in [11] by proposing linguistic constructs for defining an MPL as an SPL that imports other SPLs. The feature model and the generator of the importing SPL is deeply integrated with the feature models and the generators of the the imported SPLs, respectively, and so this model does not have a good support for modularity.

A formal model for SPL and MPL, with the goal of studying refinement to allow safe evolution of MPL was proposed in [**? ?**]. This model is close to ours, but less general as it enforces the generator to implement a compositional approach [1] and adds several constraints on the SPL definition to ensure sound refinement

[2] `https://bugs.gentoo.org`



(a) The `kde-frameworks` category **(b) The `gnome-base` category**

Figure 2: Two dependency sub-graphs of Portage

and correct MPL definition, but are incompatible with the structure of Portage.

Schröter et al. [23] informally discussed the challenges in designing an MPL, and identified several aspects that an SPL should expose in addition to its variability in order to help SPL composition. In particular, they discuss *syntactical interfaces* that correspond to the variable API of an SPL, and *behavioral interfaces* that describe the correct usage of this variable API. More recently, Schröter et al. [22] proposed the concept of feature model interface that consists of a subset of features and used it in combination with a concept of feature model composition to support compositional analyses of feature models. This current paper's notions of feature model *extension* and *refinement* are inspired by the concepts introduced in [22], and the different results of that work could be almost directly reused here. Additionally, ways to effectively define and implement feature model composition have been largely studied [**? ? ?**] in the context of the definition of a single SPL with a very large feature model. These works are complementary to ours, and could be very useful for designing an implementation of our MPL model.

Kästner et al. [16] proposed a variability-aware module and interface system that allows for type checking modules in isolation. Similarily to [10], this work is bound to a specific programming language and a specific SPL implementation based on `#ifdef` preprocessor directives and variable linking. Moreover, in contrast to our declarative SPL and refinement relation, module interfaces do not support hiding features and dependencies.

To the best of our knowledge, few works have studied the Package structure of Portage, and none drew a parallel between Gentoo and the notion of MPL. Zeng et al. [28] compared the graph structure that raise from the dependencies between package in Portage to complex networks, and developed two network growth models to study the evolution of Portage over time. Bloemen et al. [5] presented how the set of packages evolved in Portage over time, and in particular, they drew a picture of the dependencies in the KDE project.

7 CONCLUSION

In this paper we presented a formal model for Multi Software Product Line, based on the concept of Dependent Software Product Line and subtyping. We used this model to encode the full set of packages in the Gentoo Linux Distribution, thus showing that our model is capable of expressing the variability and the dependencies of 19486 components corresponding to 38907 Software Product lines. We then used our model to define few prototype analysis tools that extracted some information on the variability and the dependencies of these SPLs.

In future work, we plan to investigate using our model to directly create an MPL. We also would like to prove some interesting properties, like stating which conditions are enough to ensure that an MPL generator is a total function, and study how to extend existing formal analysis on this model, like feature model analysis [**?**], type checking [9, 10], model checking [25] or abstract interpretation [**?**]. Moreover, we would like to continue to work on the Gentoo Linux Distribution and we plan to ask the Gentoo community for comments and advice on our modeling experiments. In particular, we would like to investigate the possibility of defining a correct

and complete solver for our MPL model and to compare it with the Portage ad-hoc dependency solver. Similar works were already undertaken in the context of the debian package manager (were packages have no variability) [**? ?**] with good results.

REFERENCES

[1] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, Berlin, Germany.

[2] David Auber, Daniel Archambault, Romain Bourqui, Maylis Delest, Jonathan Dubois, Bruno Pinaud, Antoine Lambert, Patrick Mary, Morgan Mathiaut, and Guy Melançon. 2014. Tulip III. In *Encyclopedia of Social Network Analysis and Mining*. Springer, Berlin, Germany, 2216–2240.

[3] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *SPLC 2005 (Lecture Notes in Computer Science)*, Vol. 3714. Springer, Berlin, Germany, 7–20.

[4] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-cortÃĩs. 2007. FAMA: Tooling a framework for the automated analysis of feature models. In *In Proceeding of the First International Workshop on Variability Modelling of Softwareintensive Systems (VAMOS) (Lero Technical Report)*, Vol. 2007-01. 129–134.

[5] Remco Bloemen, Chintan Amrit, Stefan Kuhlmann, and Gonzalo Ordóñez Matamoros. 2014. Gentoo Package Dependencies over Time. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, New York, NY, USA, 404–407.

[6] P. Clements and L. Northrop. 2001. *Software Product Lines: Practices & Patterns*. Addison Wesley Longman, Boston, USA.

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 238–252.

[8] Patrick Cousot and Radhia Cousot. 2014. Abstract Interpretation: Past, Present and Future. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, New York, NY, USA, 2:1–2:10.

[9] Ferruccio Damiani and Michael Lienhardt. 2016. On Type Checking Delta-Oriented Product Lines. In *IFM 2016 (LNCS)*, Vol. 9681. Springer, Berlin, Germany, 47–62.

[10] Ferruccio Damiani, Michael Lienhardt, and Luca Paolini. 2017. *A Formal Model for Multi SPLs*. Springer, Berlin, Germany, 67–83.

[11] Ferruccio Damiani, Ina Schaefer, and Tim Winkelmann. 2014. Delta-oriented Multi Software Product Lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*. ACM, New York, NY, USA, 232–236.

[12] A. Dvurečenskij and S. Pulmannová. 2000. *New Trends in Quantum Structures*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

[13] Gentoo Foundation. 2017. `https://gentoo.org`. (2017).

[14] Gentoo Foundation. 2017. `https://wiki.gentoo.org/wiki/Portage`. (2017).

[15] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A systematic review and an expert survey on capabilities supporting multi product lines. *Information & Software Technology* 54, 8 (2012), 828–852. https://doi.org/10.1016/j.infsof.2012.02.002

[16] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-aware Module System. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, New York, NY, USA, 773–792.

[17] Michael Lienhardt. 2017. `https://github.com/HyVar/gentoo_to_mspl/blob/translator/PORTAGE.md`. (2017).

[18] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, Pittsburgh, PA, USA, 231–240.

[19] K. Pohl, G. Böckle, and F. van der Linden. 2005. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, Berlin, Germany.

[20] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond (SPLC 2010) (LNCS)*, Vol. 6287. SPringer, Berlin, Germany, 77–91.

[21] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495. https://doi.org/10.1007/s10009-012-0253-y

[22] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *38th International Conference on Software Engineering (ICSE)*. ACM, New York, USA, 667–678.

[23] Reimar Schröter, Norbert Siegmund, and Thomas Thüm. 2013. Towards Modular Analysis of Multi Product Lines. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*. ACM, New York, NY, USA, 96–99.

[24] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. ACM, New York, NY, USA, 63–71.

[25] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. Family-Based Model Checking with mCRL2. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Vol. 10202. Springer, Berlin, Germany, 387–405. https://doi.org/10.1007/978-3-662-54494-5_23

[26] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 1–45.

[27] Alexander Wilce. 1998. Perspectivity and congruence in partial abelian semigroups. *Mathematica Slovaca* 48, 2 (1998), 117–135.

[28] Xiaolong Zheng, Daniel Zeng, Huiqian Li, and Feiyue Wang. 2008. Analyzing open-source software systems as complex networks. *Physica A: Statistical Mechanics and its Applications* 387, 24 (2008), 6190 – 6200.